The Thesis Committee for Varsha Bhaskar

certifies that this is the approved version of the following Thesis:

Optimizing Dynamic Reseeding of Tests in a UVM Testbench

Committee:

Jacob Abraham, Supervisor

Mark McDermott

Optimizing Dynamic Reseeding of Tests in a UVM Testbench

by

Varsha Bhaskar

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

The University of Texas at Austin

May 2018

Dedicated to my Family

Optimizing Dynamic Reseeding of Tests in a UVM Testbench

Varsha Bhaskar, M.S.E

The University of Texas at Austin, 2018

Supervisor: Jacob Abraham

The coverage problem has been a long standing issue in simulation based verification. Coverage metrics are required to track the progress and justify completeness of simulation vectors. This thesis basically describes the importance of seeding the UVM tests to obtain the necessary coverage. It presents a technique of optimized dynamic reseeding of the UVM tests to meet the coverage requirement. This thesis may be called an extension to [1] following the same structure with a parallel interface to get dynamic information from the the UVM testbench to the evaluation loop which does the reseeding procedure. The process of figuring out the next best seed has been optimized with various techniques leading to faster dynamic reseeding. The optimization algorithms have been applied to two UVM testbenches. The first is a very simplified UVM testbench of a comparator from [2]. The second is an UVM testbench with an elaborate test suite that has been constructed for an AXI

AMBA 4 Interconnect. The DUT from [2] was chosen to show the improvements caused by optimization techniques applied in this thesis in the reseed evaluation with respect to the reseeding method in [1] thus highlighting the improvement caused by optimization in this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The ever increasing size and the complexity of modern day designs have posed great challenges to the current day verification. It is believed that about 75 % of the total development time goes to verification. The verification space is ever expanding, thus leading to an explosion in the number of covergroups and points written for any single feature in the design. Even though formal verification gives a comprehensive coverage for the given assumptions, the state space explosion and numerous false failures still remains a major concern leading to the adoption of UVM simulation based verification by most of the verification clusters out in the industry. UVM based verification also comes with an extra advantage of easy scalability and test porting across various designs. The main steps for an UVM based test regression are test generation and output evaluation. The test generation step mainly runs the various sequences on the DUT through a virtual interface. The output is evaluated through assertion failures and coverage obtained through covergroups and cover properties. The latter step of output evaluation is static. There is a suspense until the test completes in the regression to figure out whether all the necessary input values have

been hit by the randomly selected seed. In other words, we need to wait till the completion of the test to figure out if the necessary covergroups have been hit. This thesis puts an end to this suspense by guiding the test dynamically. This is done by changing the seeds dynamically through out the UVM test run to reach the needed coverage value through the covergroups in a single run. Chapter 2 describes coverage in detail. The next paragraph briefly describes an UVM testbench and Chapter 3 modifies the UVM testbench for dynamic reseeding described in the Thesis.

A typical UVM testbench is constructed in the following way for a DUT. To interact with the DUT and test its functionality, there exists a block sequencer that generates sequences to be sent over to the DUT by a driver block via an interface. The monitor block samples the inputs and the outputs of the DUT, tries to make a prediction of the expected result and sends the prediction and result from the DUT to another block, the scoreboard. The scoreboard compares and evaluates the behaviour of the DUT. This block usually contains some system verilog assertions to test the correct behaviour of the DUT. This block is also responsible for measuring coverage through covergroups and coverproperties. Sequencers, drivers and monitors compose an agent. An agent and a scoreboard compose an environment. All these blocks are controlled by a greater block called test. The test and DUT are encapsulated in a block called top. Figure 1.1 from [24] shows a typical UVM testbench.

This thesis puts checkpoints throughout the UVM test simulation for various coverage objectives. The best possible seed is selected by backtracking to the previous check point until a good seed has been discovered for the current coverage objective jump. A parallel interface called RSEED_INTERFACE has been constructed in the UVM environment to obtain the current coverage value on the targeted cov-

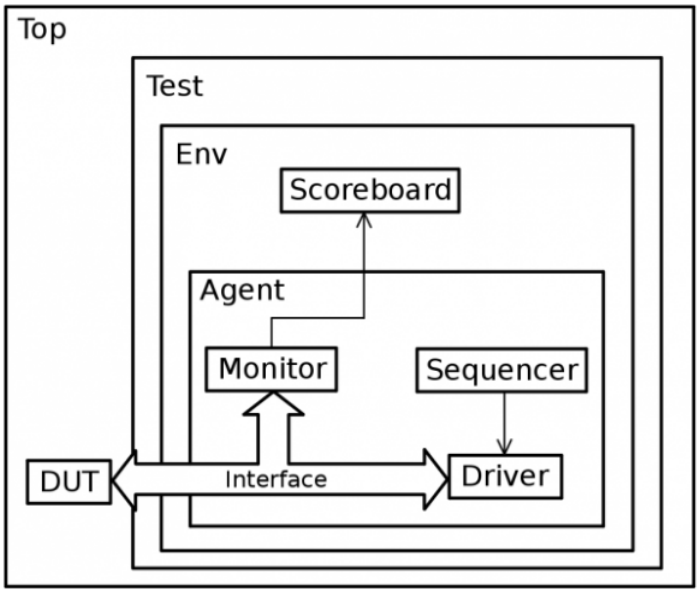

Figure 1.1: Typical UVM Testbench

ergroup. This obtained coverage value and its respective seed is evaluated in a seed evaluation loop written in TCL. The structure of obtaining values from the UVM testbench through the RSEED_INTERFACE and the concept of check pointing is similar to what has been presented in [1]. However the backtracking is done cleverly with several optimization techniques similar to simulated annealing and hill climbing with random reset to avoid local minima problems which is a major issue in [1]. This improves the number of iteration counts needed for determining the next best seed and hence reduces the time required to find the correct seed pattern for the suspense problem described in the previous paragraph.

The problem of dynamic reseeding is non-convex in nature. Through the dynamic reseeding procedure in [1], the author is able to get the coverage values dynamically from the UVM testbench using a parallel interface and checkpoint suc-

cessfully throughout the UVM test simulation without any major overhead. However, he uses convex optimization techniques in the evaluation loops which makes his solutions a victim to the local minimas. This thesis is an attempt to optimize the entire dynamic reseeding procedure by finding the lowest number of iteration cycles needed to obtain the necessary coverage objectives by cleverly avoiding the local minimas. As stated above, simulated annealing has been applied as the major optimization technique. Simulated Annealing is an optimization method that is often used to help find a global optimum amidst many local minimas for a particular function or problem. A detailed description of the work done in [1] in given in Chapter 3. Chapter 4 describes all the optimization techniques used in thesis.

The thesis also presents some mutation techniques to try and determine similarities, if any between adjacent seed numbers like 2 and 3 or seeds with their bits flipped etc; there seems to be no evident relation between any of the seeds. However results of dynamic reseeding have been presented with some specific mutation on seeds for future research if any. An attempt can be made to select a population of seeds with a fitness function to optimize dynamic reseeding with genetic mutation.

To sum up, this chapter describes the need for dynamic reseeding. It gives a brief description of the work done in [1]. It also describes the basic skeleton used for applying the optimization with a brief introduction to the techniques used in optimization. A detailed explanation of entire procedure will be given in the subsequent chapters.

Chapter 2

Overview of Coverage, Covergroups and Coverpoints

In simple terms, coverage is a metric used to measure how exhaustively a DUT has been verified. This might be in terms of the lines of code covered or on functionality or on some condition like branches taken, toggles in bits and so on. Coverage models are important to a simulation environment as they are used to measure progress and ensure completeness in the testing of the DUT. This chapter describes some major cover metrics that are used widely. It also dives deep into the functional coverage metric and gives a deeper understanding of covergroups and coverpoints.

2.1 Code Coverage Metrics

Coverage metrics can be broadly classified into two major categories, code coverage and functional coverage metrics as defined in [3]. Code coverage metrics focus on ensuring that the implementation of the design (RTL) is thoroughly exercised by the simulation vectors. Some code coverage metrics are line coverage, branch coverage

and toggle coverage. The basic code coverage metrics as defined in [3],[4],[5],[6] have been elaborated below:

1. Line/Block/Statement coverage - These metrics report whether or not every line of the RTL have been executed. It is one of the most widely used code coverage metric to measure coverage.

2. Branch/Decision coverage - This metric targets the control flow and checks if each decision outcome of every branch in the RTL has been tested.

3. Path coverage - This metric reports the number of times every possible path in the RTL was executed

4. Toggle coverage - Toggle coverage reports if every bit of a register or a wire has toggled its value.

5. Condition coverage - Condition coverage tests whether all the permutations of the variables in the condition have occurred. As an Example $X \| Y \| Z$ must be tested with all the 8 combinations of X,Y,Z.

These are some of the commonly defined code coverage metrics. Each of them have their own advantages and drawbacks. A high Line and Branch Coverage does not ensure the exhaustive testing of all the functionality of the DUT. As stated in the previous chapter, the ever increasing DUT sizes are leading to exponential number of paths. Hence it is unrealistic to rely on only on Path and Line coverage. Toggle and Condition coverage are indirect ways to support functional coverage.

2.2 Functional Coverage Metrics

The objective of Functional Coverage Metrics is to measure the verification progress with respect to functional requirements of the design. Functional Coverage has been very instrumental in today's Constraint Random Verification approach which uses a large stimulus to verify the DUT. Given the large number of tests and the traces generated by them, it is impossible to manually debug through test reports and waveforms to extract the coverage.

Since functional coverage is not an implicit coverage metric, it cannot be extracted automatically. A functional coverage model needs to be created to extract the functional coverage information from any design.

The functional behavior of any design consists of data and temporal components. Hence, there are two types of functional coverage metrics we need to consider: covergroups and Cover Properties. In [7] covergroups and Cover Properties are defined as the following:

1. Functional Data Coverage: This form of functional coverage is called covergroup modelling. It consists of state values sampled at a particular time from buses interfaces or registers . A covergroup generally has the following constructs as enumerated in [8]:

   (a) Clocking Event : Defines the event at which coverage points are sampled. If the clocking event is omitted, users must procedurally trigger the coverage sampling.

   (b) Cover Points : A cover point can be an integral variable or an integral expression. Each cover point includes a set of bins associated with its sampled values or its value transitions. The bins can be explicitly defined

by the user or automatically created by System Verilog.

(c) Cross Coverage : Coverage group can also specify cross coverage between two or more coverage points or variables.

(d) Coverage Options : These are used to control the behaviour of the covergroup.

An example of a covergroup with the features are shown below:

```
covergroup memory @ (posedge ce);
        data_op : coverpoint opr_rw {
        bins write    = 1;
        bins read     = 0;
        }
        data_add : coverpoint data {
        bins low      = {0,50};
        bins med      = {51,150};
        bins high     = {151,255};
        }
        add_op : cross data_op ,data_add;
endgroup
```

In this example the coverpoints bin the occurrences of read and write to their corresponding memory locations independently in points data_op and data_add. The cross coverage point add_op measures whether read and write happens in all regions of the memory. All the sampling of data happens at the positive edge of signal ce.

2. Functional Time Coverage: This method of coverage ensures that every par-

ticular sequence of events is properly tested. We use cover property modeling to measure temporal relationships between sequences of events. The cover properties are usually implemented with assertions.

2.3 Use of Coverage Metrics

Other than the usage of coverage metrics mentioned above, here are some others. [9], [10], [11] show methods of coverage directed test generation. The use of data mining on coverage data to assist in test generation has been presented in [12]. [13] uses a coverage guided mining algorithm for generating assertions for a design.

2.4 Final Words

In this chapter, we established the coverage problem and looked at different methods for coverage measurement. This chapter also describes the most commonly used coverage metrics and their uses. This Thesis uses System Verilog Command get_coverage() on the covergroups and their corresponding coverpoints to determine coverage dynamically as the simulation progresses. The future chapters explain the method of using cover points in dynamic reseeding and its optimization techniques.

Chapter 3

Dynamic UVM Test Seeding

This Chapter describes the procedure used in reseeding the UVM tests. It gives the motivation and benefits of using dynamic seeding as compared to static seeding of UVM tests. The chapter extends the application from [1] to a UVM testbench with components like multiple interfaces, agents, monitors, drivers, scoreboard, etc.

The procedure introduces a concept called Objective Function. An Objective Function is defined as the value obtained by calling the covergroup built-in function get_coverage() on a particular covergroup or adding the resultant get_coverage() values from a number of related covergroups. The Objective Function returns a number that increases in value as it approaches its goal. Checkpointing is the second requirement to implement this method of Constrained Random Dynamic Reseeding. This procedure primarily applies an initial seed to the UVM testbench. This seed runs for a fixed time interval called iteration time. The obtained objective function is sent to an evaluation loop via a parallel interface from the UVM testbench. The evaluation loop written in TCL(Transactional Control Language) compares the increase in the Objective Function to the value obtained at the previous checkpoint.

If the Objective Function has not increased over the time interval, the simulation time is moved back to the previous checkpoint to restart the iteration loop with a new seed. This new seed is generated in the evaluation loop and sent back to the UVM testbench via the same parallel interface. The procedure for generating the new seed in described later. If the seed resulted in an improvement to the Objective Function, then simulation time continues using the latest seed which caused an improvement in the objective function for the next iteration time interval. The seeds that caused an improvement in the objective function are stored in a log and is used to regenerate the simulation with this particular order of seeds. As the evaluation loop is written in TCL, it will be addressed as TCL Evaluation loop from now on in this thesis.

3.1 Motivation

The major motivation for this procedure is a paper written by Tom Murphy VII that was presented in 2013 [20] and [1] presented in DVCON 2017. In his paper, Murphy teaches his Computer how to play a Nintendo game by just observing the RAM locations from time to time. He does not try to understand what is going on with the game mechanics or even look at the output video or audio. He looks at the registers during every frame while the game is running and develops algorithms to infer whether it was progressing or not without knowing what the goal of the game really was. The following lines briefly describes his algorithm. An objective function is deduced from the player's inputs. This objective is then used to guide the search over all the possible inputs using an emulator. This generalizes the notion of progress alowing the algorithm to produce novel game play. The objective function is made as elegant as possible to demonstrate that the game is reducible to such

a simple objective function. Optimization methods are then used to maximize the objective function which causes the player to achieve a higher score for the given input scenarios.

The approach here is similar. There is no need to have any detailed information about the design. All that is needed is the value of the objective function, time to time from the UVM testbench. The progress in simulation is inferred through the increase in the objective function alone. Artificial Intelligence has not been applied anywhere in this procedure. This process of just taking a difference of coverage at various checkpoints makes the overhead really small, easy to implement and port it across designs.

3.2 Benefits

One of the benefits about the overhead time and portability was already brought out in the previous section. Here are some other benefits of this method. The biggest advantage of this method is that it requires absolutely no knowledge about the design. Another advantage of this procedure as compared to [1] is that it is not susceptible to local minima problems. On realizing the non convex nature of the problem, the evaluation loops have been written with optimization to avoid local minima issues.

The final advantage for this procedure as stated from [1] is its parallelism to Formal Property Verification. This procedure finds out an optimum seed tree for the given objective function. We can define an objective function for each of the individual tests. We can keep applying the optimized reseeding procedure until we reach that expected objective value thus making it comprehensive in coverage measurement. This comprehensiveness in coverage makes it similar to Formal Property

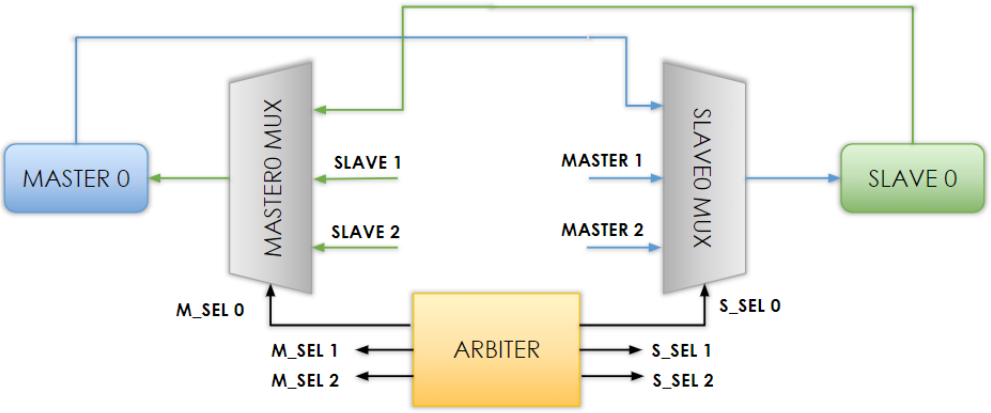

Figure 3.1: Schematic of AXI Interconnect

verification.

3.3 Dynamic Reseeding of the AXI Interconnect

This section explains how the the architecture as specified in [1] has been applied to a fairly complicated UVM testbench of an AXI Interconnect. The RTL and its corresponding UVM testbench for the AXI AMBA4 interconnect has been developed for the purposes of testing out the procedure described in this thesis. Figure 3.1 is the schematic diagram of the AXI Interconnect. The interconnect follows the protocol in [21].

The interconnect supports up to 3 masters and 3 slaves. It has a 32 bit address bus and a 64 bit data bus. It has a separate read and write arbiter and is capable determining invalid slave address error conditions. The arbiter supports dynamic priority arbitration. Slaves are selected based on the address range requested by the master. During any case of contention during arbitration, the master with the

highest priority is granted access and then moved to the lowest priority to prevent starvation.

M0	S1	1
M1	S2	2
M2	S0	3

M0	S1	3
M1	S2	1
M2	S0	2

M0	S0	2
M1	S1	3
M2	S2	1

Figure 3.2: Arbitration Scheme of the Interconnect

Figure 3.2 shows the arbitration scheme in detail. Here, the three tables denote three consecutive iterations. In the first table M0 is at the highest priority and the grant is given to M0. In the next iteration M0 is moved to the lowest priority and M1 is given grant with M1 is at the highest priority. Similarly M1 is at the lowest priority in the last iteration and M2 is at the highest priority. This shows that this arbitration scheme prevents starvation. The same arbitration scheme is followed for read and write.

Figure 3.3 shows the modified UVM testbench constructed for doing the dynamic reseeding procedure. The main inclusions as compared to a conventional UVM testbench are the RSEED_INTERFACE and the TCL evaluation loops. Since this design is way more complex than the one in [1] changes have been made to the UVM test, UVM sequence and the UVM sequence item from the existing conventional test bench for an AXI INTERCONNECT.

RSEED_INTERFACE provides the simulator a known path to call System Verilog functions that dynamically interrogate or modify the verification environment. This interface acts as an intermediary between the TCL evaluation function and the System Verilog functions from the UVM testbench. Values are changed

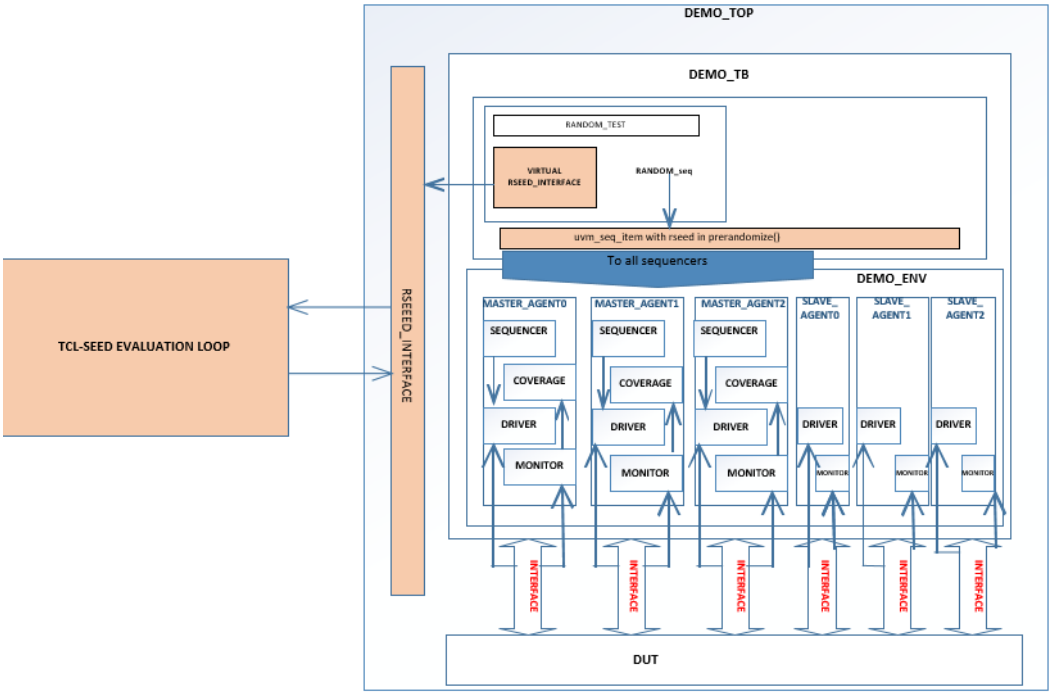

Figure 3.3: UVM Testbench of AXI INTERCONNECT modified for Dynamic Reseeding

dynamically using the this parallel interface using the TCL commands.

RSEED_INTERFACE also defines the variables needed for triggering, checkpointing and measuring coverage dynamically. The interface has definitions for the functions which set seed and set coverage to a particular calculated value. These functions are called by TCL commands in the evaluation loop to set seed and obtain dynamic coverage for evaluation. The coverage values from the covergroups which are under considerations are also measured here using the get_coverage() command on the covergroup. Lastly the parallel interface also grabs values from the DUT at time 0 and initializes the TCL simulator with those values. The figures 3.4a

and 3.4b below shows a snippet of RSEED_INTERFACE obtaining coverage using get_coverage() from a coverpoint in a covergroup.

```
covergroup objective_cg;
        coverpoint match1;
        coverpoint match2;
        coverpoint match3;
endgroup
```

(a) Covergroup

```
coverage_value = AXI_interconnect.objective.match1.get_coverage() + AXI_interconnect.objective.match2.get_coverage();
```

(b) Coverage Measurement

Figure 3.4: Use of get_coverage()

The uvm_sequence_item consists of data fields required for generating the stimulus in the sequence of an UVM test. To generate the stimulus, the sequence items are randomized in sequences. Therefore, data properties which are used in the stimulus generation of the UVM tests should be declared as rand and can have constraints defined. For successful reseeding, the pre-randomize phase of the uvm_sequence_item does the work of purging an existing seed if one exists in the seed table. The seed_table ensures that an attempt is made to randomize every object in a deterministic way based of some intrinsic properties of the UVM object. The method described allows the built-in UVM method reseed() to function as we want; a method to dynamically reseed an object. It is of at most importance that we remove the existing seed entry before we seed because the the reseed() function looks for any extant seed database before creating a new hash.

The constructed UVM testbench for the AXI Interconnect has a test suite with simple tests such as read followed by write by a particular master and slave, parallel reads and writes between 2 masters and slaves and contention for the same slave for read or write by the same Master. There is one other test which is a random mixture of all these simple tests with 3 Masters and 3 slaves with the application

of dynamic priority arbitration. The code snippet in figures 3.5 and 3.6 shows this random test named rand_test before and after modification for this procedure.

```
    .........
    endfunction : new

    virtual function void build_phase(uvm_phase phase);
    ..........
        seq0 = rand_seq::type_id::create("seq0");
        seq1 = rand_seq::type_id::create("seq1");
        seq2 = rand_seq::type_id::create("seq2");

        phase.raise_objection(this);

        //Master 0,1,2 interacting randomly with different slaves

        assert( seq0.randomize() with {seq0.num_req==num_req1;seq0.delay_min==delay_min1;seq0.delay_max==delay_max1;} );
        assert( seq1.randomize() with {seq1.num_req==num_req1;seq1.delay_min==delay_min1;seq1.delay_max==delay_max1;} );
        assert( seq2.randomize() with {seq2.num_req==num_req1;seq2.delay_min==delay_min1;seq2.delay_max==delay_max1;} );
        fork
        seq0.start(m_demo_tb.m_axi_env.m_masters[0].m_sequencer);
        seq1.start(m_demo_tb.m_axi_env.m_masters[1].m_sequencer);
        seq2.start(m_demo_tb.m_axi_env.m_masters[2].m_sequencer);
        join

        phase.drop_objection(this);

 endtask: run_phase

endclass : random_test
```

Figure 3.5: Random Test without Modification

These are the modifications that are to be added to any existing UVM tests and testbenches to support the specified method of dynamic seeding for obtaining the required coverage.

The most important block of this procedure is the TCL evaluation loop. The TCL evaluation loop in [1] does the following steps to get the right sequence of seeds:

1. Start the test sequence with a random seed specified using +ntb_random_seed argument from the command line.

2. The seed runs in the simulator for a fixed interval time called iteration time. If the coverage obtained is greater than the value obtained at the previous check point then step 3 is performed. If the coverage is equal to or greater than the

```
class random_test extends Demo_base_test;

  //{initialization of some test variables}
   .......
   virtual rseed_interface rseed_interface;
   .......
   if (!uvm_config_db#(virtual rseed_interface)::get(this, "", "rseed_interface", rseed_interface)) begin
         `uvm_fatal("test0", "Failed to get rseed_interface")
    end

    endfunction : new

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
        seq0 = rand_seq::type_id::create("seq0");
        seq1 = rand_seq::type_id::create("seq1");
        seq2 = rand_seq::type_id::create("seq2");

  endfunction : build_phase

  task run_phase(uvm_phase phase);
  phase.raise_objection(this);
        ..........
        fork
                begin
                        forever begin
`
        assert{ seq0.randomize() with {seq0.num_req==num_req1;seq0.delay_min==delay_min1;seq0.delay_max==delay_max1;} };
        assert( seq1.randomize() with {seq1.num_req==num_req1;seq1.delay_min==delay_min1;seq1.delay_max==delay_max1;} );
        assert( seq2.randomize() with {seq2.num_req==num_req1;seq2.delay_min==delay_min1;seq2.delay_max==delay_max1;} );
                        fork
                        seq0.start(m_demo_tb.m_axi_env.m_masters[0].m_sequencer);//0,1,2

                        seq1.start(m_demo_tb.m_axi_env.m_masters[2].m_sequencer);
                        seq2.start(m_demo_tb.m_axi_env.m_masters[1].m_sequencer);
                        join

                         #(700);
                        end
                end
                wait (rseed_interface.final_report == 1);
                join_any


   phase.drop_objection(this);

  endtask: run_phase

virtual function void end_of_elaboration_phase (uvm_phase phase);
        super.end_of_elaboration_phase (phase);
        uvm_top.set_timeout (40000ns);
    endfunction

endclass : random_test
```

Figure 3.6: Random Test with Modification

final objective then step 4 is performed. If neither of the above mentioned cases are true , then the simulation jumps to step 5.

3. The current seed is accepted for the next iteration time period of the simulation

and step 2 is performed again. The checkpoint is updated to the current simulation time with an increased next objective function.

4. If the final objective is reached, the simulation stops after giving the correct seed sequence as the output with the number of iterations that was required to reach the objective.

5. If there was no increase in the coverage objective as compared to the previous checkpoint in the iteration time, then the simulation rewinds back to the previous checkpoint and selects another random seed through the TCL command rand() and sets it in the UVM testbench using the set_seed() function from RSEED_INTERFACE and the simulation returns to step 2.

3.4 Observations on the Procedure of Dynamic Seeding

From section 3.3, it is evident that the reseeding procedure is an optimization algorithm in its core to find a solution to obtain the objective coverage. This problem is clearly not convex in nature. The method of just comparing whether the next seed has an increase in coverage as compared to the previous seed makes this procedure a victim of the local minima problem. The solution for the problem of attaining the given objective coverage can be obtained in lesser number of cycles. This can be achieved by using some optimization techniques in the TCL evaluation loop. The solutions to the problem might still not be the most ideal one. However, the optimization can still bring down the iteration cycle time to half its value in some of the cases.

This thesis tackles this problem going forward. It implements the techniques similar to Hill Climbing Random Reset [15] and Simulated Annealing [17] in the TCL

evaluation loop, and shows the difference in the reduction of iteration cycles achieved as compared to the current implementation in [1]. It compares the values with the DUT from [2] and the AXI Interconnect UVM testbench which was constructed throughout this chapter.

Chapter 4

Optimization

This chapter first defines what are Convex and Non convex problems in the context of optimization and then goes on to explain why the problem here is non convex and how the hill climbing optimization applied in [1] is susceptible to local minima problems. The Chapter explains the optimization methods of Hill Climbing with random restart and Simulated Annealing in detail. It explains how these methods have been slightly modified for this problem statement in the thesis to get an optimized list of seeds to obtain the required objective coverage.

4.1 Types of Optimization Problems

Optimization problems can broadly be divided into two categories Convex and Non Convex Optimization problems.

4.1.1 Convex Optimization Problem

A convex optimization problem is a problem where all of the constraints are convex functions, and the objective is a convex function if minimizing, or a concave

function if maximizing. In a convex optimization problem, the feasible region is the intersection of the convex constraint functions. An example for a convex function is given in Figure 4.1 from [14].

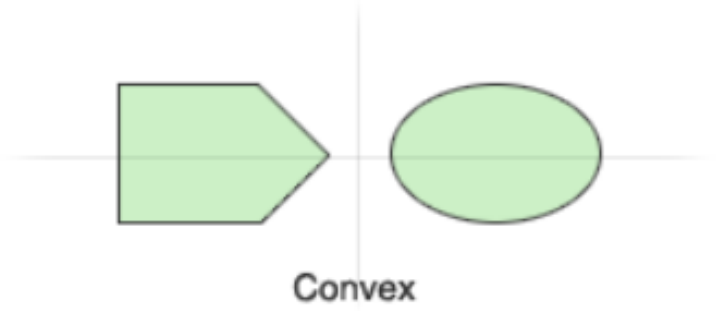

Figure 4.1: Convex Function

With a convex objective and a convex feasible region, there can be only one optimal solution, which is globally optimal. There are many algorithms such as hill climbing [22] which are very powerful in finding the global optima for convex functions.

4.1.2 Non Convex Optimization Problem

A non-convex optimization problem is any problem where the objective or any of the constraints are non-convex. A non convex optimization problem is pictured in Figure 4.2 from [14]. Such problems may have multiple feasible regions and multiple locally optimal points within each region. When one uses techniques of optimization that are used in convex optimization problems, such solutions are susceptible to be stuck in local minima issues.

After the explanations given above for convex and non convex functions, it is evident that the procedure to find the set of seeds needed to attain the given objec-

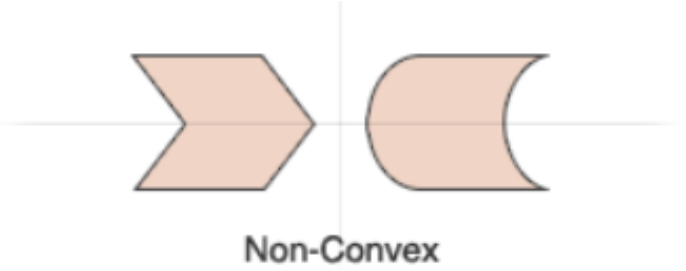

Figure 4.2: Non Convex Function

tive through dynamic reseeding is non convex in nature. Hence better optimization techniques are to be applied to get to the the most optimized seed path.

The TCL evaluation loop in [1] has the hill climbing optimization procedure to determine the right order of the seeds. Hill Climbing is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. Here the initial seed is chosen at random during the start of the simulation. The method described in section 3.3 is followed which is similar to the Hill Climbing Optimization method.

The term seed tree has been used a few times before this in this thesis. Seed tree is the tree of seeds produced due to reseeding from the Hill Climbing procedure. If there is a totally new seed produced not due to to the Hill Climbing optimization procedure, but due to the new optimization procedures introduced, then it is called a new seed tree. This has been demonstrated in the figure 4.3 shown in the next page.

Figure 4.3: Seed Tree

4.2 Hill Climbing Random Reset

The first attempt to solve the the local minima problem was to tweak the existing algorithm. The tweaking is done to produce an optimization technique similar to one of the varieties of Hill climbing with random reset. The application is similar to the work done in [15] and [16].

The current application of the optimization algorithm can be called Hill Climbing with random reseed. The simulation is allowed to start with a random seed as before. The general hill climbing optimization procedure is allowed to run for a fixed number of iteration cycles. Then regardless of the current state of simulation

there is a reseed from the current seed tree to a completely new seed tree. This reseeding happens regardless of the increment or decrements in the coverage during the iteration cycles.

This random reseeding helps the simulation to get out of the local minimas if it is stuck. The results for this procedure for various values of iterations, cover points and DUTs are given in detail in Chapter 5. The advantage of using this random reseed in shown in Figure 4.4. This randomization technique produces some positive results in terms of reduction in the number of iterations like the one shown in the figure. The random reseeding after cycle 3 as shown in the figure helps the simulation to move to a better tree with lesser iteration cycles taken to reach the final objective.

However, this optimization method has its own drawbacks of wedges and valleys. If the reseeding always jumps to such a seed tree where there is no increase in coverage for a large number of cycles, then this optimization method might end up taking more time than the conventional hill climbing method. Such cases can also be observed in the results shown in Chapter 5. Figure 4.5 shows such a case.

This example shows that random reseeding without considering the objective value of the previous seed, might not lead to a correct seed tree always as shown in the previous example in Figure 4.5. Hence we need to compare two seed trees before jumping from one to another. There can be only certain calculated jumps to seed tree with lesser coverage values at that current time to remove the simulation from local minima. This brings us to the next method : Simulated Annealing.

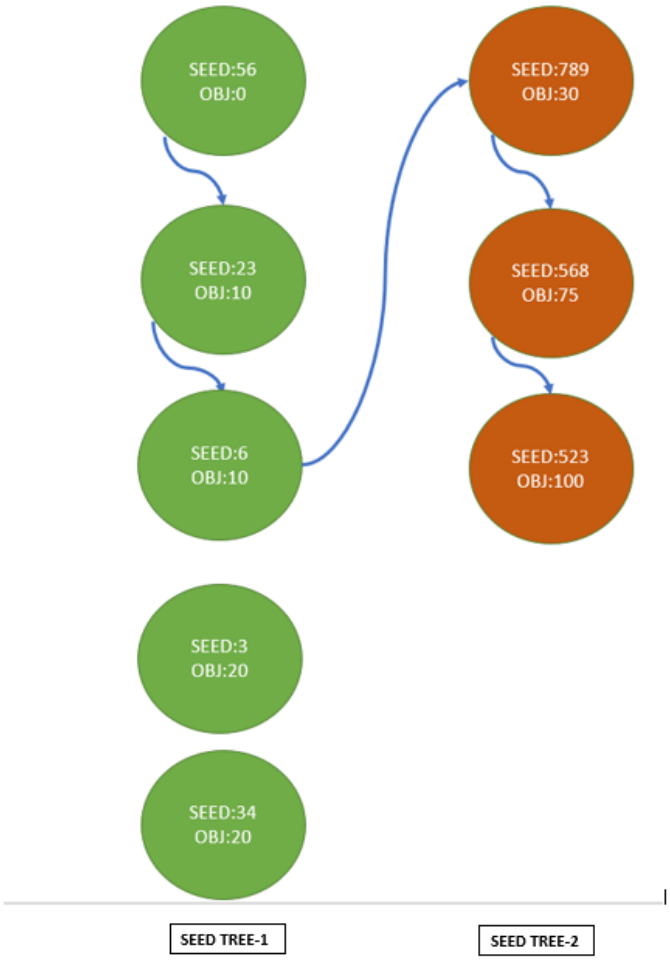

Figure 4.4: Hill Climbing Example-1

4.3 Simulated Annealing

Simulated Annealing from [19] is defined as a general form of optimization useful in finding global optima in the presence of large numbers of local optima. Annealing refers to an analogy with thermodynamics, specifically with the way that metals cool and anneal. Simulated annealing uses the objective function of an optimization

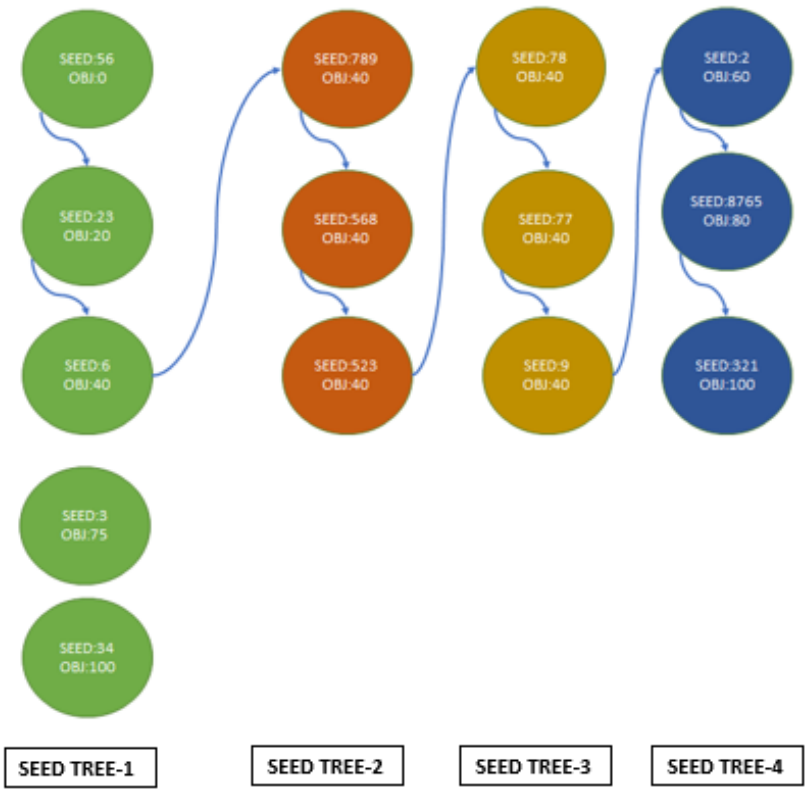

Figure 4.5: Hill Climbing Example-2

problem instead of the energy of a material.

The name and inspiration of this technique comes from Metallurgy. As per [17], the technique involves heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. Both are attributes of the material that depend on its thermodynamic free energy. Heating and cooling the material affects both the temperature and the thermodynamic free energy. The simulation of annealing is used to find an approximation of a global minimum for a function

with a large number of variables to the statistical mechanics of annealing of the mathematically equivalent artificial multiatomic system.

Broadly, an optimization algorithm searches for the best solution by generating a random initial solution and "exploring" the area nearby. If a neighboring solution is better than the current one, then it moves to it. If not, then the algorithm stays put. Simulated annealing injects just the right amount of randomness to escape local maxima early in the process without getting off course late in the game, when a solution is nearby. This makes it pretty good at tracking down a decent answer, no matter what its starting point is. The figure 4.6 from [18] shows the exact case where simulated annealing is helpful.

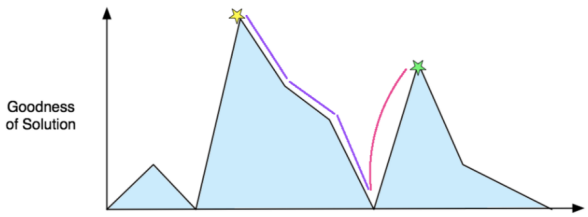

Figure 4.6: Avoiding of Local Minima by Simulated Annealing

The hill climbing as applied in [1] has a possibility to get stuck in the green star in the its search for the number of iterations to get the right seed order to reach the objective function. Simulated Annealing tries to get to the right slope of increment as early as possible in the simulation.

From [19] the basic algorithm for Simulated Annealing :

1. Generate a random solution.

2. Calculate the cost function/objective function.

3. Generate a neighbouring random solution.

4. Calculate the cost function from the new solution.

5. Compare the solution

   - If cost_new <cost_old: move to the new solution.
   - If cost_new >cost_old: move to the new solution only based on a probability.

6. Repeat steps 3-5 above until an acceptable solution is found or you reach some maximum number of iterations.

Elaborating on the second bullet of Step 5 in the above algorithm: The algorithm will eschew moving to a worse solution. However doing it all the time would make it get caught in the local minima. To avoid this problem, it sometimes elects to keep the worse solution. To decide, the algorithm calculates something called the 'acceptance probability' which is a number between 0 and 1 and then compares it to a random number between 0 and 1 generated through the random number generation command in that any programming language like TCL, C, PERL, etc.

4.4 Application of Simulated Annealing in Dynamic Reseeding

This section explains the how simulated annealing is applied to the current problem. The most expensive process in terms of time in this procedure is the Evaluation Loop which outputs the list of order of seeds to be used in the simulation with the number of iterations taken to achieve the given objective. The number of iterations or in simple terms the *time* is the parameter we are trying to optimize using this

optimization algorithm. The modified Simulated Annealing Optimization as applied to this procedure is described below:

1. Select a random seed to start the simulation.

2. Run the hill climbing optimization procedure for a fixed number of iterations called cycle_count. Save the set of seeds generated for each iteration cycle. Calculate the coverage difference after the iteration cycle runs for cycle_count. This will be called cov_old. This set of iteration cycles is called the first set of iteration.

3. Run the simulated annealing process starting with the same seed as in the beginning of step 2. Use the same set of seeds until a pre-fixed number of iteration cycles given by rand_count.

4. Generate a new random seed when the number of simulated annealing iterations reach rand_count in number and let the iteration cycles continue till cycle_count iterations with seeds generated based on the hill climbing procedure. The coverage difference is calculated after the cycle_count number of iteration cycle runs for simulated annealing. This difference is called cov_sa. The set of iterations run in steps 3 and 4 will be called the second set of iterations.

    - If cov_sa >cov_old: the seed for the next iteration is the last seed obtained after the second set of simulated annealing iterations. The simulated annealing procedure has discovered a better seed tree.
    - If cov_sa <cov_old: move to the new solution only based on a probability.

5. If the solution after second set of iterations is accepted, then the seed for the next iterations is the last seed after the second set of iterations. However, if the first set of iteration is considered as the better seed tree, the next iteration will have the last seed as saved from the first set of iterations.

6. Repeat steps 2-5 until you reach the objective function.

The flowchart for the procedure has been drawn in figure 4.7

As described in the procedure for Simulated Annealing, the acceptance probability is defined as the following:

$$acceptance\_probability = \frac{cov\_old - cov\_sa}{maximum\_objective}$$

The acceptance_probability is then compared with a random number between 0 and 1. If the acceptance_probability is greater than the generated random number then the new solution is accepted. As the coverage objective progresses through the simulation, then both cycle_count and rand_count are increased. This ensures that the jumps are more initially to avoid the local minima and reseeds less due to the simulated annealing procedure as the simulation moves towards completion.

The results for the optimization methods described in this chapter are presented in the next chapter for 2 DUTs. It will be observed that simulated annealing produces a shorter number of total iteration cycles for most of the cases.

A stint of Genetic Mutation was also tried on the seeds to try to obtain an optimized solution if one existed. The results for this procedure are also presented in the next Chapter.

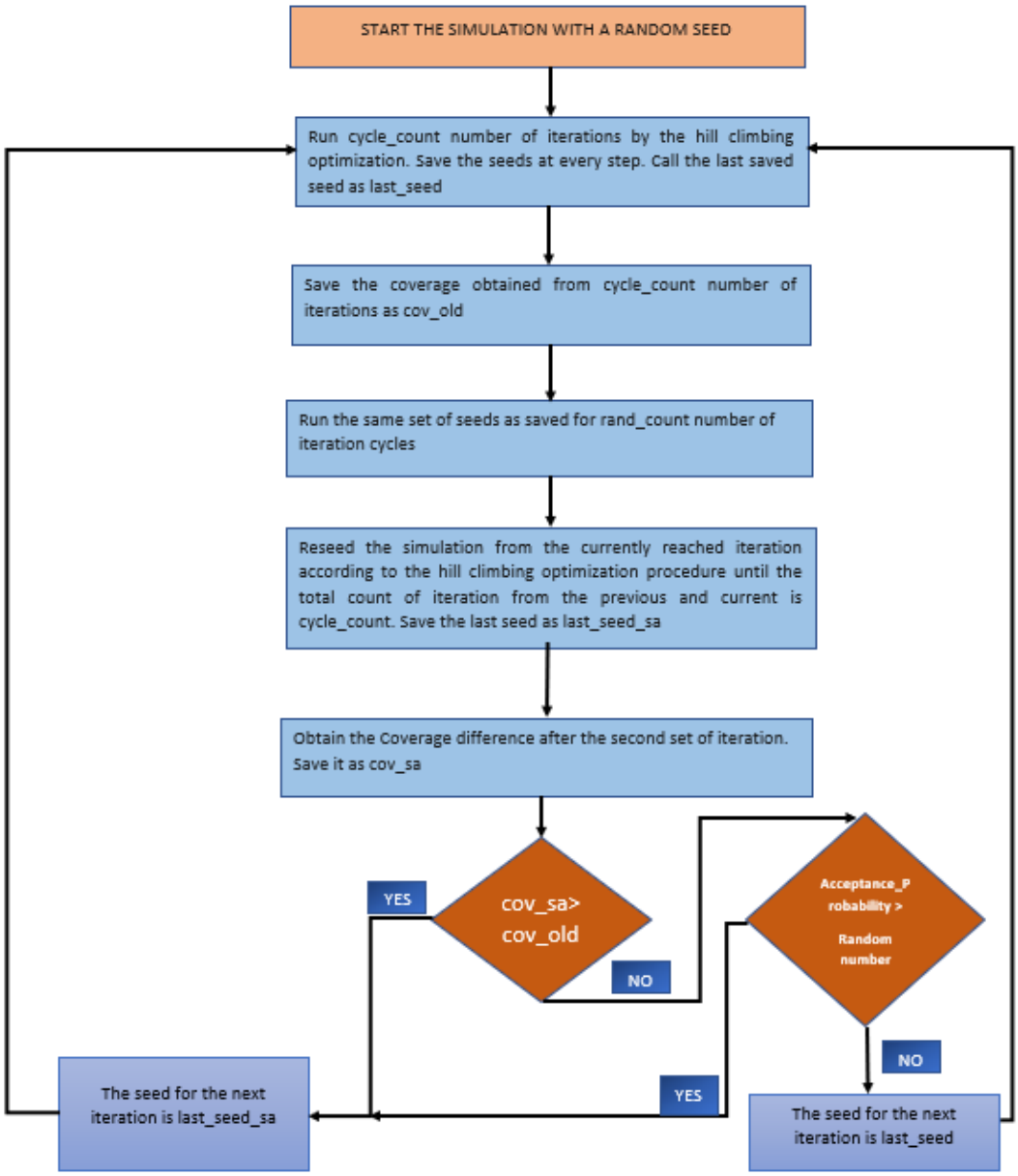

Figure 4.7: Algorithm for Simulated Annealing

Chapter 5

Results

As mentioned in the previous chapters, two UVM testbenches have been chosen to demonstrate the results. The covergroups have been made for those values which are the most important part of the driver phase in the UVM testbench developed for both the DUTs under consideration.

5.1 DUT-1

The first UVM testbench and DUT is a design based on the comparison between two values as used in [2]. The snippet of the code has been given below in figure 5.1 using which we define EX1 and EX2:

From figure 5.1 EX1 shows that the variable c can be 1 only if a equals b. EX2 captures coverage when variable b has 1s in all its bits for all values of a. The terms EX1 and EX2 will have the same meaning as described in the previous sentence from here on in this thesis. For EX1, 4 out 16 possibilities tell us that (a==b). The probability that these combinations will come in back to back iterations are $\frac{4!}{(2^4)^4}$. It takes 33 iterations to find all the 4 possibilities using the traditional hill

```
module dut #(parameter width=2) (
       input [width-1:0] a,
       input [width-1:0] b,
       input           clk,
       input           reset,
       output          c
                                 );

   reg [width-1:0]           match;

   // grab coverage automatically
   covergroup objective_cg;
      coverpoint match;
   endgroup

   objective_cg objective;

   // EX 1. DEFAULT a == b match
   //assign c = (a == b);

   // EX2. b is all 1s cross with all values of a
    assign c  = (&b);

   // assign match to value of a
   always @(posedge clk) begin
      if (c) begin
         match <= a;
         objective.sample();
      end else begin
         match <= '0;
      end


   end

   // initialize covergroup
   initial begin
      objective = new();
   end
```

Figure 5.1: DUT-1

climbing method. However after optimization through hill climbing with random reseed and simulated annealing produce results in 17 and 10 iterations respectively. The iteration cycles may differ in number from machine to machine. However, the

number of cycles will have a similar relative ratio.

The results in the following pages shows the comparison in the number of iterations obtained from hill climbing with random reseed and simulated annealing for various initial seeds, iteration time duration and a number of different widths to show the performance of the optimization algorithm. It can be observed from the results that hill climbing with random reseed works well in some cases and in some cases it does not. Simulated annealing works well for most of the cases as observed in the 12 result tables for EX1 and EX2. The iteration cycle time duration also plays an important role in the performance of the optimization procedure. The effect of iteration cycle time can be observed from tables 5.1 and 5.2.

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
81	51	27	3
148	123	46	4
837	792	412	5
timeout	timeout	1534	7

Table 5.1: No. of iterations with 50ns iteration time for DUT-1 EX1 seed : 76545

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
20	9	19	3
155	148	132	4
667	581	609	5
timeout	timeout	1564	7

Table 5.2: No. of iterations with 60ns iteration time for DUT-1 EX1 seed: 785468

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
51	81	27	3
204	221	98	4
634	914	592	5
timeout	timeout	1539	7

Table 5.3: No. of iterations with 50ns iteration time for DUT-1 EX1 seed : 687

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
81	33	29	3
185	187	184	4
502	936	482	5
timeout	timeout	1674	7

Table 5.4: No. of iterations with 50ns iteration time for DUT-1 EX1 seed : 4576

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
51	34	35	3
187	184	164	4
752	822	679	5
timeout	timeout	1574	7

Table 5.5: No. of iterations with 50ns iteration time for DUT-1 EX1 seed : 64

5.2 DUT-2 : AXI INTERCONNECT

The other UVM testbench under consideration is that of the AXI Interconnect which was constructed throughout Chapter 3. All the three optimization techniques described till now have been applied to the UVM testbench one by one. The cover-

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
57	60	19	3
275	168	131	4
706	1235	653	5
timeout	timeout	timeout	7

Table 5.6: No. of iterations with 50ns iteration time for DUT-1 EX1 seed : 98

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
27	21	16	3
275	168	134	4
789	1106	749	5
timeout	timeout	timeout	7

Table 5.7: No. of iterations with 50ns iteration time for DUT-1 EX1 seed : 3821

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
44	24	34	3
214	211	193	4
907	904	798	5
timeout	timeout	timeout	7

Table 5.8: No. of iterations with 50ns iteration time for DUT-1 EX1 seed : 1642339558

group that has been selected is the one that bins the lengths of the transactions. The transaction lengths for both read or write can have lengths from 1-8, 1-16, 1-32 and are selected randomly for each transaction in the rand_test for either read or write. The parameter rand_count defined in Chapter 4 has been modified in table

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
37	61	31	3
235	169	89	4
1157	929	529	5
timeout	timeout	1534	6

Table 5.9: No. of iterations with 50ns iteration time for DUT-1 EX2 seed : 1296041802

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
30	69	45	3
346	411	279	4
timeout	975	1156	5
timeout	timeout	timeout	6

Table 5.10: No. of iterations with 40ns iteration time for DUT-1 EX2 seed : 905546

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
40	16	18	3
181	146	131	4
972	1089	700	5
timeout	timeout	timeout	6

Table 5.11: No.of iterations with 50ns iteration time for DUT-1 EX2 seed : 304

5.13 and its results have been reported. Since the priorities of the masters change dynamically, the interval time has been chosen in such a way that three operations read or write can be completed in one interval time. From table 5.13, it is shown that the point of interruption and the number of cycles after which the seeds change

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Width of A&B
3	3	3	2
37	40	22	3
197	159	127	4
905	1365	567	5

Table 5.12: No. of iterations with 50ns iteration time for DUT-1 EX2 seed : 2

due to the induced randomness also affect the performance of simulated annealing.

Length	Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	rand_count
8	19	43	21	2
8	19	43	22	4
8	19	43	13	6
16	73	102	74	2
16	73	102	67	4
16	73	102	54	6
16	73	102	58	8
32	293	259	170	2
32	293	259	166	4
32	293	259	193	6
32	293	259	255	8

Table 5.13: No. of iterations with 1000ns iteration time for DUT-2 seed : 8734

Table 5.14 shows how simulated annealing performs with different start seeds for a length of 32 with a rand_count of 6.

It is observed that simulated annealing also gives similar number of iteration cycles regardless of the starting seed which is not observed in the other two optimization evaluation loops. Table 5.15 shows how simulated annealing works when

Seed	Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing
89	233	159	225
698	436	268	123
5903	976	434	136
164565	387	567	167
1676	199	223	150
5298	73	102	158
90876	344	292	175
3	953	759	140
3290	674	176	169
50945	200	297	163

Table 5.14: No. of iterations with 1000ns iteration time for DUT-2

the objective has coverpoints from 2 covergroups as shown in the code snippets in Figure 5.2. Simulated annealing gives a similar result for dynamic reseeding for this combination of covergroups as well.

The objective function is the addition of the coverage from the coverpoints of separate covergroups. The equation for the same is assigned in the variable coverage_value as shown below:

coverage_value= AXI_interconnect.objective.match1.get_coverage() +

AXI_interconnect.objective.match2.get_coverage();

All these results show that the heart of dynamic reseeding procedure, the TCL evaluation loop is optimized by applying simulated annealing. It helps us to find the right order of seeds faster than the method suggested in [1]. This has been checked for different types of covergroups and in every case the dynamic reseed table or seed tree is constructed faster than hill climbing method of evaluation in the TCL loops. The reason for this optimization in the number of iterations and its consistency across various seeds and covergroups is due to the fact that the UVM

```
covergroup objective_cg;
        coverpoint match1;
endgroup

covergroup objective_cg1;
        covergroup match2;
endgroup

objective_cg objective;
objective_cg1 objective1;

assign c = (AWLEN_M2<='d64 || AWLEN_M3 <= 'd64 );

always @(posedge ACLK) begin
        if(c) begin
                match1<=(AWLEN_M1);
                match2<=(AWLEN_M2);
                match3<=(AWLEN_M3);
                objective.sample();
                objective1.sample();
        end
        else begin
                match1<='d0;
                match2<='d0;

        end
end

 // initialize covergroup
   initial begin
        objective = new();
        objective1 = new();
   end
```

Figure 5.2: Combination of Cover groups

test eschews away from the local minima as early as possible in the UVM test run procedure. Thus this system of dynamic reseeding of UVM tests performs better with the applied simulated annealing.

For regressions, where one would like to set a specific objective for each test, different command line arguments can be passed for each test. This selection is unique and must trigger the objective associated with that particular test only. The RSEED_INTERFACE described in Chapter 3 can have the corresponding map

Seed	Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing
89	201	185	184
3145967290	308	468	201
2308355234	273	223	199
873663295	254	310	189
164565	209	197	155
225250845	569	250	173
1417617288	290	262	175
4040396367	265	269	184
1403213963	893	264	258
9	267	202	192
580572021	timeout	219	193

Table 5.15: No. of iterations with 1000ns iteration time for DUT-2 : objective 200

between the variable arguments from the command line and the objective function specific to each test. This ensures that each test achieves its objective, thus reducing the time spent in aimless verification.

The entire code was developed and tested with VCS 2016 on 64 bit Linux. The same technique can be extended to other simulators as well. The evaluation loops can also be written in PERL or Python.

5.3 Mutation

An effort was made to find a relation between the consecutive seed numbers or between the seed number and its exact inverse in terms of bits (eg: 0x0001 & 0xFFFE). One attempt was to mutate the successful seed number to the seed number incremented by 1. This method is abbreviated as MT-1. Another attempt was to generate a seed which was completely the inversion of all the bits of a successful

seed. The method is called MT-2. The results with the consecutive seeds are given in tables 5.16, 5.17, 5.18. The results with the bits flipped are given in tables 5.19, 5.20, 5.21.

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Genetic Mutation	Width of A&B
81	51	27	30	3
148	123	46	143	4
837	792	412	1146	5
timeout	timeout	1534	timeout	7

Table 5.16: No. of iterations with MT-1 50ns iteration time for DUT-1 EX1 seed : 76545

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Genetic Mutation	Width of A&B
30	69	45	47	3
346	411	279	163	4
timeout	975	1156	639	5
timeout	timeout	timeout	timeout	6

Table 5.17: No. of iterations with MT-1 40ns iteration time for DUT-1 EX2 seed : 905546

No improvement was found with respect to the simulated annealing method of optimization. There was no consistency in time for the results obtained across a number of seeds as well. In some cases the same set of consecutive seed numbers behave differently at various time instances in the same UVM test run. This

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Genetic Mutation	Width of A&B
57	60	19	49	3
275	168	131	209	4
706	1235	653	700	5
timeout	timeout	timeout	timeout	7

Table 5.18: No. of iterations with MT-1 50ns iteration time for DUT-1 EX1 seed : 98

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Genetic Mutation	Width of A&B
57	60	19	30	3
275	168	131	144	4
706	1235	653	930	5
timeout	timeout	timeout	timeout	7

Table 5.19: No. of iterations with MT-2 50ns iteration time for DUT-1 EX1 seed : 98

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Genetic Mutation	Width of A&B
30	69	45	36	3
346	411	279	143	4
timeout	975	1156	655	5
timeout	timeout	timeout	timeout	6

Table 5.20: No. of iterations with MT-2 40ns iteration time for DUT-1 EX2 seed: 905546

Hill Climbing	Hill Climbing with Random Reseed	Simulated Annealing	Genetic Mutation	Width of A&B
81	51	27	32	3
148	123	46	142	4
837	792	412	844	5
timeout	timeout	1534	timeout	7

Table 5.21: No. of iterations with MT-2 50ns iteration time for DUT-1 EX1 seed : 76545

shows that the generation of random variables through consecutive UVM seeds is completely random in nature.

5.4 Future Work

This procedure of dynamic reseeding can be implemented with parallelization. Multiple seeds can be run in parallel. Each of these parallel paths can implement simulated annealing and the one particular seed with maximum objective may be selected after the interval time through the parallel seed paths. If the simulation is controlled by a C code or a VIP [23] and does not obey the check pointing, then the simulation state will not be consistent when a checkpoint is restored and the simulation will be invalid. There can be some relation developed either in C or TCL to remove this invalidity.

The iteration interval time has to be decided with care for each of the UVM tests. There can be some work done to generalize it across the designs. A truly genetic algorithm can be applied for optimization where one can start with a population of seeds and use a fitness function to select an improved set of seeds. There can

be other optimization procedures which may perform better than the ones suggested in the thesis.

Bibliography

[1] E.Nelson.Improving Constrained Random Testing by Achieving Simulation Verification Goals through Objective Functions, Rewinding and Dynamic Seed Manipulation In Proceeding of Design and Verification Conference (DVCON), 2017

[2] E.Nelson,"dut.sv".,https://github.com/tenthousandfailures/improving constrained-random/blob/master.

[3] Coverage cookbook http://verificationacademy.com/cookbook/coverage.

[4] Jing-Yang Jou and C Liu. Coverage analysis techniques for hdl design validation. In Proceedings of Asia Pacific Chip Design Languages, pages 48-55, 1999.

[5] Andrew Piziali. Functional verification coverage measurement and analysis. Springer Science & Business Media, 2007.

[6] Serdar Taziran and Kurt Keutzer. Coverage metrics for functional validation of hardware designs. IEEE Design & Test of Computers, 18(4):36- 45, 2001.

[7] Universal Verification Methodology. http://www.learnuvmverification.com /index.php/category/functional-coverage.

[8] http://www.asic-world.com/systemverilog/coverage2.html

[9] Mike Benjamin, Daniel Geist, Alan Hartman, Yaron Wolfsthal, Gerard Mas, and Ralph Smeets. A study in coverage-driven test generation. In Design Automation Conference, 1999. Proceedings. 36th, pages 970-975. IEEE, 1999.

[10] Jayanta Bhadra, Magdy S Abadir, Li-C Wang, and Sandip Ray. A survey of hybrid techniques for functional verification. IEEE Design & Test of Computers,24(2):0112-122, 2007.

[11] Onur Guzey and Li-C Wang. Coverage-directed test generation through automatic constraint extraction. In High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International, pages 151-158. IEEE, 2007.

[12] Monica Farkash-presenter. Mining coverage data for test set coverage efficiency.

[13] David Sheridan, Lingyi Liu, Hyungsul Kim, and Shobha Vasudevan. A coverage guided mining approach for automatic generation of succinct assertions. In Proceedings of the 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, pages 68-73. IEEE Computer Society,2014.

[14] https://www.solver.com/convex-optimization

[15] C. F. F. C. Filho and M. G. F. Costa and J. E. C. Filho and A. L. M. de Oliveira Using a random restart hill-climbing algorithm to reduce component assembly time in printed circuit boards In Proceedings of the 2010 IEEE International Conference on Industrial Technology, pages 1706-1711.IEEE, 2010.

[16] W. Lee, S. Lee, B. Lee, Y. Lee A Genetic Optimization Approach to Operation

of a Multi Head Surface Machine In EICE Trans. Fundamentals, vol. E83-A, no. 9, pages 1748-1756, 2000.

[17] A. Khachaturyan, S. Semenovsovskaya and B. Vainshtein The thermodynamic approach to the structure analysis of crystals In Acta Crystallographica Section A pages 742–754 , 1981.

[18] http://katrinaeg.com/simulated-annealing.html

[19] S. Kirkpatrick, C. D. Gelatt Jr. and M. Vecchi Optimization by Simulated Annealing In Science, vol.220, no. 4598, pages 671-680, 1983.

[20] T. Murphy VII The First Level of Super Mario Bros. is Easy with Lexicographic Orderings and Time Travel . . . after that it gets a little tricky In Sigbovik, Pittsburgh, 2013.

[21] ARM Limited, 'AMBA AXI and ACE Protocol Specification'

[22] http://pages.cs.wisc.edu/ jerryzhu/cs540/handouts/hillclimbing.pdf

[23] https://www.design-reuse.com/wiki/verification

[24] https://colorlesscube.com/uvm-guide-for-beginners/chapter-2-defining-the-verification-environment